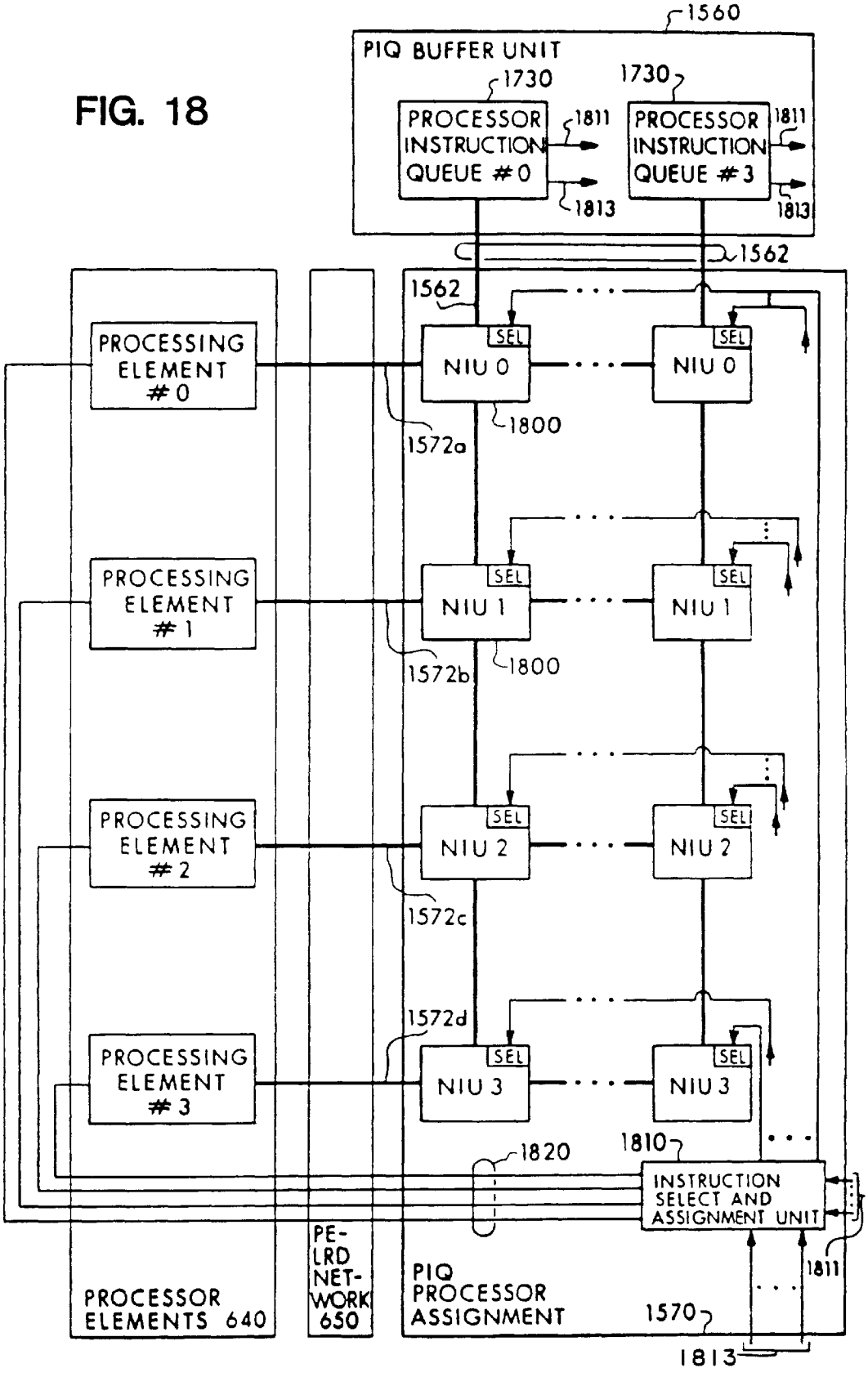
BIAX Corporation v. Intel
Civil Action No. 2:05-cv-184-TJW

EXHIBIT 4
(PART 2)
FIRST AMENDED COMPLAINT FOR PATENT INFRINGEMENT

FIG. 18


1560
PIQ BUFFER UNIT
1730
1730
PROCESSOR INSTRUCTION QUEUE #0
PROCESSOR INSTRUCTION QUEUE #3
1811
1811
1813
1813
1562
1562
PROCESSING ELEMENT #0
PROCESSING ELEMENT #1
PROCESSING ELEMENT #2
PROCESSING ELEMENT #3
SEL
NIU 0
NIU 1
NIU 2
NIU 3
1572a
1572b
1572c
1572d
1800
1820
1810
INSTRUCTION SELECT AND ASSIGNMENT UNIT
PROCESSOR ELEMENTS 640
PE-LRD NET-WORK 650
PIQ PROCESSOR ASSIGNMENT
1570
1811
1813

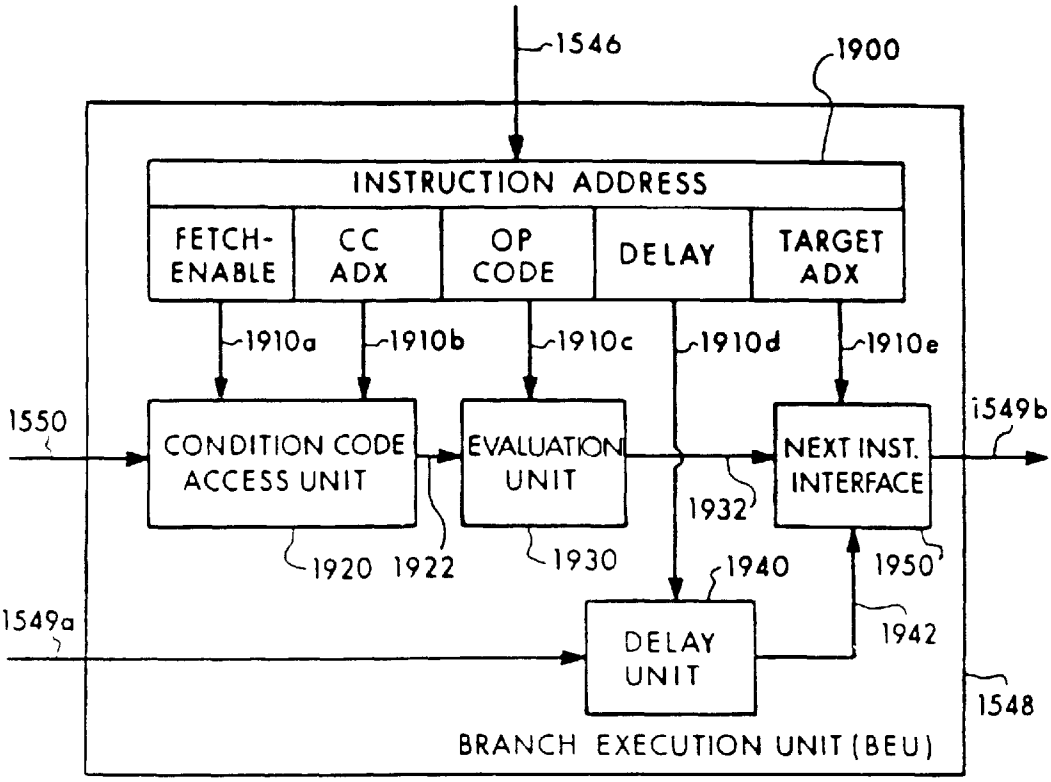

1546
1900
INSTRUCTION ADDRESS
FETCH-ENABLE
CC ADX
OP CODE
DELAY
TARGET ADX
1910a
1910b
1910c
1910d
1910e
1550
CONDITION CODE ACCESS UNIT
EVALUATION UNIT
NEXT INST. INTERFACE
1549b
1932
1920
1922
1930
1940
1950
1549a
DELAY UNIT
1942
1548
BRANCH EXECUTION UNIT (BEU)

FIG. 19

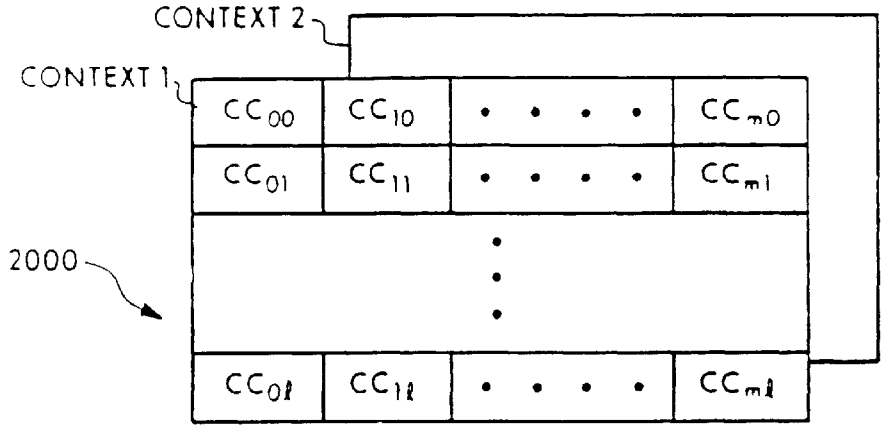

CONTEXT 2
CONTEXT 1
CC_{00}
CC_{10}
CC_{m0}
CC_{01}
CC_{11}
CC_{m1}
2000
CC_{0l}
CC_{1l}
CC_{ml}

FIG. 20

FIG. 21

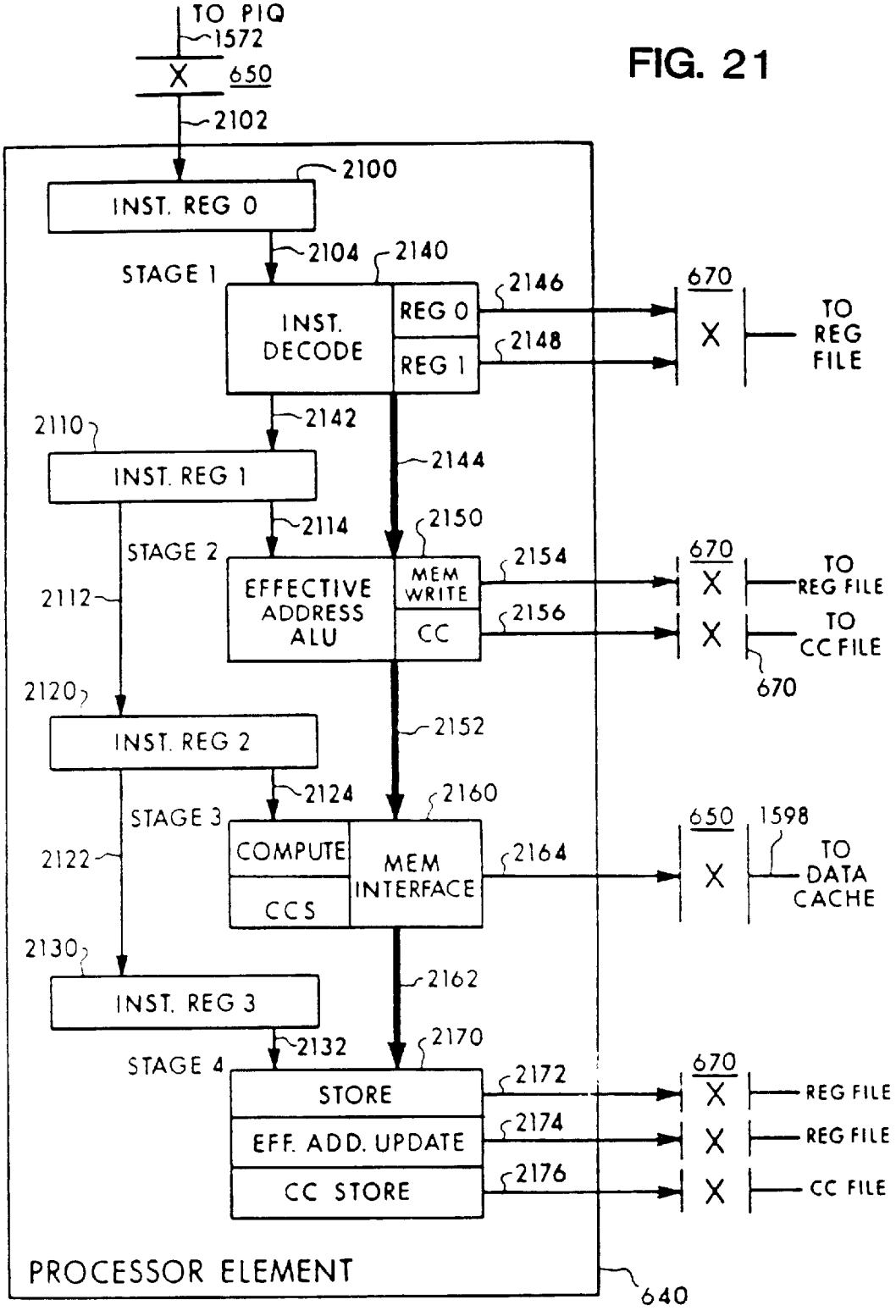

TO PIQ
1572
X
650
2102
2100
INST. REG 0
2104
2140
STAGE 1
INST. DECODE
REG 0
REG 1
2146
2148
670
X
TO REG FILE
2110
2142
INST. REG 1
2144
2114
2150
STAGE 2
2112
EFFECTIVE ADDRESS ALU
MEM WRITE
CC
2154
2156
670
X
TO REG FILE
X
TO CC FILE
670
2120
INST. REG 2
2152
2124
2160
STAGE 3
2122
COMPUTE
CCS
MEM INTERFACE
2164
650
1598
X
TO DATA CACHE
2130
INST. REG 3
2162
2132
2170
STAGE 4
STORE
EFF. ADD. UPDATE
CC STORE
2172
2174
2176
670
X
REG FILE
X
REG FILE
X
CC FILE
PROCESSOR ELEMENT
640

FIG. 22a

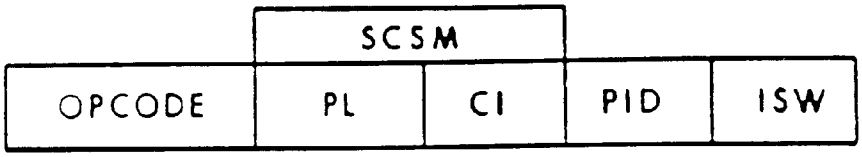


FIG. 22b

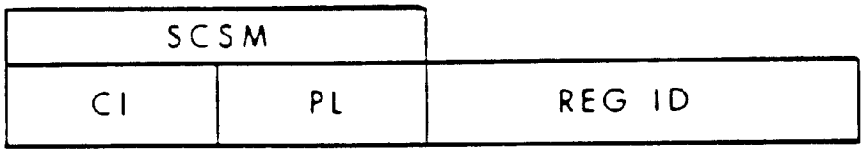


FIG. 22c

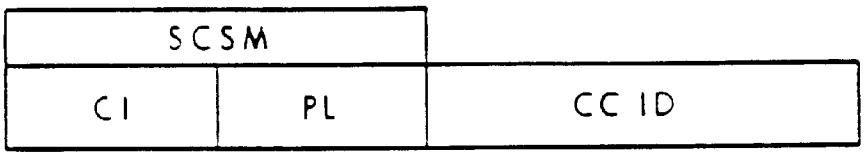
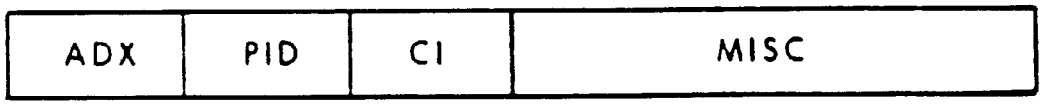


FIG. 22d



PARALLEL PROCESSOR SYSTEM FOR PROCESSING NATURAL CONCURRENCIES AND METHOD THEREFOR

This is a divisional of U.S. Ser. No. 08/254,687, filed Jun. 6, 1994, now U.S. Pat. No. 5,517,628, which is a divisional of Ser. No. 08/093,794, filed Jul. 19, 1993, now abandoned, which is a continuation of Ser. No. 07/913,736, filed Jul. 14, 1992, now abandoned, which is a continuation of Ser. No. 07/560,093, filed Jul. 30, 1990, now abandoned, which is a divisional of Ser. No. 07/372,247, filed Jun. 26, 1989, now U.S. Pat. No. 5,021,945, which is a divisional of Ser. No. 06/794,221, filed Oct. 31, 1985, now U.S. Pat. No. 4,847,755.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention generally relates to parallel processor computer systems and, more particularly, to parallel processor computer systems having software for detecting natural concurrencies in instruction streams and having a plurality of processor elements for processing the detected natural concurrencies.

2. Description of the Prior Art

Almost all prior art computer systems are of the "Von Neumann" construction. In fact, the first four generations of computers are Von Neumann machines which use a single large processor to sequentially process data. In recent years, considerable effort has been directed towards the creation of a fifth generation computer which is not of the Von Neumann type. One characteristic of the so-called fifth generation computer relates to its ability to perform parallel computation through use of a number of processor elements. With the advent of very large scale integration (VLSI) technology, the economic cost of using a number of individual processor elements becomes cost effective.

Whether or not an actual fifth generation machine has yet been constructed is subject to debate, but various features have been defined and classified. Fifth-generation machines should be capable of using multiple-instruction, multiple-data (MIMD) streams rather than simply being a single instruction, multiple-data (SIMD) system typical of fourth generation machines. The present invention is of the fifth-generation non-Von Neumann type. It is capable of using MIMD streams in single context (SC-MIMD) or in multiple context (MC-MIMD) as those terms are defined below. The present invention also finds application in the entire computer classification of single and multiple context SIMD (SC-SIMD and MC-SIMD) machines as well as single and multiple context, single-instruction, single data (SC-SISD and MC-SISD) machines.

While the design of fifth-generation computer systems is fully in a state of flux, certain categories of systems have been defined. Some workers in the field base the type of computer upon the manner in which "control" or "synchronization" of the system is performed. The control classification includes control-driven, data-driven, and reduction (or demand) driven. The control-driven system utilizes a centralized control such as a program counter or a master processor to control processing by the slave processors. An example of a control-driven machine is the Non-von-1 machine at Columbia University. In data-driven systems, control of the system results from the actual arrival of data required for processing. An example of a data-driven machine is the University of Manchester dataflow machine developed in England by Ian Watson. Reduction driven

systems control processing when the processed activity demands results to occur. An example of a reduction processor is the MAGO reduction machine being developed at the University of North Carolina, Chapel Hill. The characteristics of the non-von-1 machine, the Manchester machine, and the MAGO reduction machine are carefully discussed in Davis, "Computer Architecture," *IEEE Spectrum*, November, 1983. In comparison, data-driven and demand-driven systems are decentralized approaches whereas control-driven systems represent a centralized approach. The present invention is more properly categorized in a fourth classification which could be termed "time-driven." Like data-driven and demand-driven systems, the control system of the present invention is decentralized. However, like the control-driven system, the present invention conducts processing when an activity is ready for execution.

Most computer systems involving parallel processing concepts have proliferated from a large number of different types of computer architectures. In such cases, the unique nature of the computer architecture mandates or requires either its own processing language or substantial modification of an existing language to be adapted for use. To take advantage of the highly parallel structure of such computer architectures, the programmer is required to have an intimate knowledge of the computer architecture in order to write the necessary software. As a result, preparing programs for these machines requires substantial amounts of the users effort, money and time.

Concurrent to this activity, work has also been progressing on the creation of new software and languages, independent of a specific computer architecture, that will expose (in a more direct manner), the inherent parallelism of the computation process. However, most effort in designing supercomputers has been concentrated in developing new hardware with much less effort directed to developing new software.

Davis has speculated that the best approach to the design of a fifth-generation machine is to concentrate efforts on the mapping of the concurrent program tasks in the software onto the physical hardware resources of the computer architecture. Davis terms this approach one of "task-allocation" and touts it as being the ultimate key to successful fifth-generation architectures. He categorizes the allocation strategies into two generic types. "Static allocations" are performed once, prior to execution, whereas "dynamic allocations" are performed by the hardware whenever the program is executed or run. The present invention utilizes a static allocation strategy and provides task allocations for a given program after compilation and prior to execution. The recognition of the "task allocation" approach in the design of fifth generation machines was used by Davis in the design of his "Data-driven Machine-II" constructed at the University of Utah. In the Data-driven Machine-II, the program was compiled into a program graph that resembles the actual machine graph or architecture.

Task allocation is also referred to as "scheduling" in Gaiski et al, "Essential Issues in Multi-processor Systems," *Computer*, June, 1985. Gajski et al set forth levels of scheduling to include high level, intermediate level, and low level scheduling. The present invention is one of low-level scheduling, but it does not use conventional scheduling policies of "first-in-first-out", "round-robin", "shortest type in job-first", or "shortest-remaining-time." Gajski et al also recognize the advantage of static scheduling in that overhead costs are paid at compile time. However, Gajski et al's recognized disadvantage, with respect to static scheduling, of possible inefficiencies in guessing the run time profile of

each task is not found in the present invention. Therefore, the conventional approaches to low-level static scheduling found in the Occam language and the Bulldog compiler are not found in the software portion of the present invention. Indeed, the low-level static scheduling of the present invention provides the same type, if not better, utilization of the processors commonly seen in dynamic scheduling by the machine at run time. Furthermore, the low-level static scheduling of the present invention is performed automatically without intervention of programmers as required (for example) in the Occam language.

Davis further recognizes that communication is a critical feature in concurrent processing in that the actual physical topology of the system significantly influences the overall performance of the system.

For example, the fundamental problem found in most data-flow machines is the large amount of communication overhead in moving data between the processors. When data is moved over a bus, significant overhead, and possible degradation of the system, can result if data must contend for access to the bus. For example, the Arvind data-flow machine, referenced in Davis, utilizes an I-structure stream in order to allow the data to remain in one place which then becomes accessible by all processors. The present invention, in one aspect, teaches a method of hardware and software based upon totally coupling the hardware resources thereby significantly simplifying the communication problems inherent in systems that perform multiprocessing.

Another feature of non-Von Neumann type multiprocessor systems is the level of granularity of the parallelism being processed. Gajski et al term this "partitioning." The goal in designing a system, according to Gajski et al, is to obtain as much parallelism as possible with the lowest amount of overhead. The present invention performs concurrent processing at the lowest level available, the "per instruction" level. The present invention, in another aspect, teaches a method whereby this level of parallelism is obtainable without execution time overhead.

Despite all of the work that has been done with multiprocessor parallel machines, Davis (Id. at 99) recognizes that such software and/or hardware approaches are primarily designed for individual tasks and are not universally suitable for all types of tasks or programs as has been the hallmark with Von Neumann architectures. The present invention sets forth a computer system and method that is generally suitable for many different types of tasks since it operates on the natural concurrencies existent in the instruction stream at a very fine level of granularity.

All general purpose computer systems and many special purpose computer systems have operating systems or monitor/control programs which support the processing of multiple activities or programs. In some cases this processing occurs simultaneously; in other cases the processing alternates among the activities such that only one activity controls the processing resources at any one time. This latter case is often referred to as time sharing, time slicing, or concurrent (versus simultaneous) execution, depending on the particular computer system. Also depending on the specific system, these individual activities or programs are usually referred to as tasks, processes, or contexts. In all cases, there is a method to support the switching of control among these various programs and between the programs and the operating system, which is usually referred to as task switching, process switching, or context switching. Throughout this document, these terms are considered synonymous, and the terms context and context switching are generally used.

The present invention, therefore, pertains to a non-Von Neumann MIMD computer system capable of simultaneously operating upon many different and conventional programs by one or more different users. The natural concurrencies in each program are statically allocated, at a very fine level of granularity, and intelligence is added to the instruction stream at essentially the object code level. The added intelligence can include, for example, a logical processor number and an instruction firing time in order to provide the time-driven decentralized control for the present invention. The detection and low level scheduling of the natural concurrencies and the adding of the intelligence occurs only once for a given program, after conventional compiling of the program, without user intervention and prior to execution. The results of this static allocation are executed on a system containing a plurality of processor elements. In one embodiment of the invention, the processors are identical. The processor elements, in this illustrated embodiment, contain no execution state information from the execution of previous instructions, that is, they are context free. In addition, a plurality of context files, one for each user, are provided wherein the plurality of processor elements can access any storage resource contained in any context file through total coupling of the processor element to the shared resource during the processing of an instruction. In a preferred aspect of the present invention, no condition code or results registers are found on the individual processor elements.

SUMMARY OF INVENTION

The present invention provides a method and a system that is non-Von Neumann and one which is adaptable for use in single or multiple context SISD, SIMD, and MIMD configurations. The method and system is further operative upon a myriad of conventional programs without user intervention.

In one aspect, the present invention statically determines at a very fine level of granularity, the natural concurrencies in the basic blocks (BBs) of programs at essentially the object code level and adds intelligence to the instruction stream in each basic block to provide a time driven decentralized control. The detection and low level scheduling of the natural concurrencies and the addition of the intelligence occurs only once for a given program after conventional compiling and prior to execution. At this time, prior to program execution, the use during later execution of all instruction resources is assigned.

In another aspect, the present invention further executes the basic blocks containing the added intelligence on a system containing a plurality of processor elements each of which, in this particular embodiment, does not retain execution state information from prior operations. Hence, all processor elements in accordance with this embodiment of the invention are context free. Instructions are selected for execution based on the instruction firing time. Each processor element in this embodiment is capable of executing instructions on a per-instruction basis such that dependent instructions can execute on the same or different processor elements. A given processor element in the present invention is capable of executing an instruction from one context followed by an instruction from another context. All operating and context information necessary for processing a given instruction is then contained elsewhere in the system.

It should be noted that many alternative implementations of context free processor elements are possible. In a non-pipelined implementation each processor element is mono-

lithic and executes a single instruction to its completion prior to accepting another instruction.

In another aspect of the invention, the context free processor is a pipelined processor element, in which each instruction requires several machine instruction clock cycles to complete. In general, during each clock cycle, a new instruction enters the pipeline and a completed instruction exists the pipeline, giving an effective instruction execution time of a single instruction clock cycle. However, it is also possible to microcode some instructions to perform complicated functions requiring many machine instruction cycles. In such cases the entry of new instructions is suspended until the complex instruction completes, after which the normal instruction entry and exit sequence in each clock cycle continues. Pipelining is a standard processor implementation technique and is discussed in more detail later.

The system and method of the present invention are described in the following drawing and specification.

DESCRIPTION OF THE DRAWING

Other objects, features, and advantages of the invention will appear from the following description taken together with the drawings in which:

FIG. **1** is the generalized flow representation of the TOLL software of the present invention;

FIG. **2** is a graphic representation of a sequential series of basic blocks found within the conventional compiler output;

FIG. **3** is a graphical presentation of the extended intelligence added to each basic block according to one embodiment of the present invention;

FIG. **4** is a graphical representation showing the details of the extended intelligence added to each instruction within a given basic block according to one embodiment of the present invention;

FIG. **5** is the breakdown of the basic blocks into discrete execution sets;

FIG. **6** is a block diagram presentation of the architectural structure of apparatus according to a preperred embodiment of the present invention;

FIGS. **7***a*–**7***c* represent an illustration of the network interconnections during three successive instruction firing times;

FIGS. **8**–**11** are the flow diagrams setting forth features of the software according to one embodiment of the present invention;

FIG. **12** is a diagram describing one preferred form of the execution sets in the TOLL software:

FIG. **13** sets forth the register file organization according to a preferred embodiment of the present invention:

FIG. **14** illustrates a transfer between registers in different levels during a subroutine call;

FIG. **15** sets forth the structure of a logical resource driver (LRD) according to a preferred embodiment of the present invention;

FIG. **16** sets forth the structure of an instruction cache control and of the caches according to a preferred embodiment of the present invention;

FIG. **17** sets forth the structure of a PIQ buffer unit and a PIQ bus interface unit according to a preferred embodiment of the present invention;

FIG. **18** sets forth interconnection of processor elements through the PE-LRD network to a PIQ processor alignment circuit according to a preferred embodiment of the present invention;

FIG. **19** sets forth the structure of a branch execution unit according to a preferred embodiment of the present invention;

FIG. **20** illustrates the organization of the condition code storage of a context file according to a preferred embodiment of the present invention;

FIG. **21** sets forth the structure of one embodiment of a pipelined processor element according to the present invention; and

FIGS. **22**(*a*) through **22**(*d*) set forth the data structures used in connection with the processor element of FIG. **21**.

GENERAL DESCRIPTION

1. Introduction

In the following two sections, a general description of the software and hardware of the present invention takes place. The system of the present invention is designed based upon a unique relationship between the hardware and software components. While many prior art approaches have primarily provided for multiprocessor parallel processing based upon a new architecture design or upon unique software algorithms, the present invention is based upon a unique hardware/software relationship. The software of the present invention provides the intelligent information for the routing and synchronization of the instruction streams through the hardware. In the performance of these tasks, the software spatially and temporally manages all user accessible resources, for example, general registers, condition code storage registers, memory and stack pointers. The routing and synchronization are performed without user intervention, and do not require changes to the original source code. Additionally, the analysis of an instruction stream to provide the additional intelligent information for controlling the routing and synchronization of the instruction stream is performed only once during the program preparation process (often called "static allocation") of a given piece of software, and is not performed during execution (often called "dynamic allocation") as is found in some conventional prior art approaches. The analysis effected according to the invention is hardware dependent, is performed on the object code output from conventional compilers, and advantageously, is therefore programming language independent.

In other words, the software, according to the invention, maps the object code program onto the hardware of the system so that it executes more efficiently than is typical of prior art systems. Thus the software must handle all hardware idiosyncrasies and their effects on execution of the program instructions stream. For example, the software must accommodate, when necessary, processor elements which are either monolithic single cycle or pipelined.

2. General Software Description

Referring to FIG. **1**, the software of the present invention, generally termed "TOLL," is located in a computer processing system **160**. Processing system **160** operates on a standard compiler output **100** which is typically object code or an intermediate object code such as "p-code." The output of a conventional compiler is a sequential stream of object code instructions hereinafter referred to as the instruction stream. Conventional language processors typically perform the following functions in generating the sequential instruction stream:

1. lexical scan of the input text,
2. syntactical scan of the condensed input text including symbol table construction,
3. performance of machine independent optimization including parallelism detection and vectorization, and

4. an intermediate (PSEUDO) code generation taking into account instruction functionality, resources required, and hardware structural properties.

In the creation of the sequential instruction stream, the conventional compiler creates a series of basic blocks (BBs) which are single entry single exit (SESE) groups of contiguous instructions. See, for example, Alfred v. Aho and Jeffery D. Ullman, *Principles of Compiler Design*, Addison Wesley, 1979, pg. 6, 409, 412–413 and David Gries, *Compiler Construction for Digital Computers*, Wiley, 1971. The conventional compiler, although it utilizes basic block information in the performance of its tasks, provides an output stream of sequential instructions without any basic block designations. The TOLL software, in this illustrated embodiment of the present invention, is designed to operate on the formed basic blocks (BBs) which are created within a conventional compiler. In each of the conventional SESE basic blocks there is exactly one branch (at the end of the block) and there are no control dependencies. The only relevant dependencies within the block are those between the resources required by the instructions.

The output of the compiler **100** in the basic block format is illustrated in FIG. **2**. Referring to FIG. **1**, the TOLL software **110** of the present invention being processed in the computer **160** performs three basic determining functions on the compiler output **100**. These functions are to analyze the resource usage of the instructions **120**, extend intelligence for each instruction in each basic block **130**, and to build execution sets composed of one or more basic blocks **140**. The resulting output of these three basic functions **120**, **130**, and **140** from processor **160** is the TOLL software output **150** of the present invention.

As noted above, the TOLL software of the present invention operates on a compiler output **100** only once and without user intervention. Therefore, for any given program, the TOLL software need operate on the compiler output **100** only once.

The functions **120**, **130**, **140** of the TOLL software **110** are, for example, to analyze the instruction stream in each basic block for natural concurrencies, to perform a translation of the instruction stream onto the actual hardware system of the present invention, to alleviate any hardware induced idiosyncrasies that may result from the translation process, and to encode the resulting instruction stream into an actual machine language to be used with the hardware of the present invention. The TOLL software **110** performs these functions by analyzing the instruction stream and then assigning processor elements and resources as a result thereof. In one particular embodiment, the processors are context free. The TOLL software **110** provides the "synchronization" of the overall system by, for example, assigning appropriate firing times to each instruction in the output instruction stream.

Instructions can be dependent on one another in a variety of ways although there are only three basic types of dependencies. First, there are procedural dependencies due to the actual structure of the instruction stream; that is, instructions may follow one another in other than a sequential order due to branches, jumps, etc. Second, operational dependencies are due to the finite number of hardware elements present in the system. These hardware elements include the general registers, condition code storage, stack pointers, processor elements, and memory. Thus if two instructions are to execute in parallel, they must not require the same hardware element unless they are both reading that element (provided of course, that the element is capable of being read simultaneously). Finally, there are data dependencies between instructions in the instruction stream. This form of dependency will be discussed at length later and is particularly important if the processor elements include pipelined processors. Within a basic block, however, only data and operational dependencies are present.

The TOLL software **110** must maintain the proper execution of a program. Thus, the TOLL software must assure that the code output **150**, which represents instructions which will execute in parallel, generates the same results as those of the original serial code. To do this, the code **150** must access the resources in the same relative sequence as the serial code for instructions that are dependent on one another; that is, the relative ordering must be satisfied. However, independent sets of instructions may be effectively executed out of sequence.

In Table 1 is set forth an example of a SESE basic block representing the inner loop of a matrix multiply routine. While, this example will be used throughout this specification, the teachings of the present invention are applicable to any instruction stream. Referring to Table 1, the instruction designation is set forth in the right hand column and a conventional object code functional representation, for this basic block, is represented in the left hand column.

TABLE 1

OBJECT CODE	INSTRUCTION
LD R0, (R10) +	I0
LD R1, (R11) +	I1
MM R0, R1, R2	I2
ADD R2, R3, R3	I3
DEC R4	I4
BRNZR LOOP	I5

The instruction stream contained within the SESE basic block set forth in Table 1 performs the following functions. In instruction **I0**, register **R0** is loaded with the contents of memory whose address is contained in **R10**. The instruction shown above increments the contents of **R10** after the address has been fetched from **R10**. The same statement can be made for instruction **I1**, with the exception that register **R1** is loaded and register **R11** is incremented. Instruction **I2** causes the contents of registers **R0** and **R1** to be multiplied and the result is stored in register **R2**. In instruction **I3**, the contents of register **R2** and register **R3** are added and the result is stored in register **R3**. in instruction **I4**, register **R4** is decremented. Instructions **I2**, **I3** and **I4** also generate a set of condition codes that reflect the status of their respective execution. In instruction **I5**, the contents of register **R4** are indirectly tested for zero (via the condition codes generated by instruction **I4**). A branch occurs if the decrement operation produced a non-zero value; otherwise execution proceeds with the first instruction of the next basic block.

Referring to FIG. **1**, the first function performed by the TOLL software **110** is to analyze the resource usage of the instructions. In the illustrated example, these are instructions **I0** through **I5** of Table I. The TOLL software **110** thus analyzes each instruction to ascertain the resource requirements of the instruction.

This analysis is important in determining whether or not any resources are shared by any instructions and, therefore, whether or not the instructions are independent of one another. Clearly, mutually independent instructions can be executed in parallel and are termed "naturally concurrent." Instructions that are independent can be executed in parallel and do not rely on one another for any information nor do they share any hardware resources in other than a read only manner.

On the other hand, instructions that are dependent on one another can be formed into a set wherein each instruction in the set is dependent on every other instruction in that set. The dependency may not be direct. The set can be described by the instructions within the set, or conversely, by the resources used by the instructions in the set. Instructions within different sets are completely independent of one another, that is, there are no resources shared by the sets. Hence, the sets are independent of one another.

In the example of Table 1, the TOLL software will determine that there are two independent sets of dependent instructions:

Set 1: CC**1**: **I0**, **I1**, **I2**, **I3**

Set 2: CC**2**: **I4**, **I5**

As can be seen, instructions **I4** and **I5** are independent of instructions **I0**–**I3**. In set 2, **I5** is directly dependent on **I4**. In set 1, **I2** is directly dependent on **I0** and **I1**. Instruction **I3** is directly dependent on **I2** and indirectly dependent on **I0** and **I1**.

The TOLL software of the present invention detects these independent sets of dependent instructions and assigns a condition code group of designation(s), such as CC**1** and CC**2**, to each set. This avoids the operational dependency that would occur if only one group or set of condition codes were available to the instruction stream.

In other words, the results of the execution of instructions **I0** and **I1** are needed for the execution of instruction **I2**. Similarly, the results of the execution of instruction **I2** are needed for the execution of instruction **I3**. In performing this analyses, the TOLL software **110** determines if an instruction will perform a read and/or a write to a resource. This functionality is termed the resource requirement analysis of the instruction stream.

It should be noted that, unlike the teachings of the prior art, the present invention teaches that it is not necessary for dependent instructions to execute on the same processor element. The determination of dependencies is needed only to determine condition code sets and to determine instruction firing times, as will be described later. The present invention can execute dependent instructions on different processor elements, in one illustrated embodiment, because of the context free nature of the processor elements and the total coupling of the processor elements to the shared resources, such as the register files, as will also be described below.

The results of the analysis stage **120**, for the example set forth in Table 1, are set forth in Table 2.

TABLE 2

INSTRUCTION	FUNCTION
I0	Memory Read, Reg. Write, Reg. Read & Write
I1	Memory Read, Reg. Write, Reg. Read & write
I2	Two Reg. Reads, Reg. Write, Set Cond. Code (Set #1)
I3	Two Reg. Reads, Reg. Write, Set Cond. Code (Set #1)
I4	Read Reg., Reg. Write, Set Cond. Code (Set #2)
I5	Read Cond. Code (Set #2)

In Table 2, for instructions **I0** and **I1**, a register is read and written followed by a memory read (at a distinct address), followed by a register write. Likewise, condition code writes and register reads and writes occur for instructions **I2** through **I4**. Finally, instruction **I5** is a simple read of a condition code storage register and a resulting branch or loop.

The second step or pass **130** through the SESE basic block **100** is to add or extend intelligence to each instruction within the basic block. In the preferred embodiment of the invention, this is the assignment of an instruction's execution time relative to the execution times of the other instructions in the stream, the assignment of a processor number on which the instruction is to execute and the assignment of any so-called static shared context storage mapping information that may be needed by the instruction.

In order to assign the firing time to an instruction, the temporal usage of each resource required by the instruction must be considered. In the illustrated embodiment, the temporal usage of each resource is characterized by a "free time" and a "load time." The free time is the last time the resource was read or written by an instruction. The load time is the last time the resource was modified by an instruction. If an instruction is going to modify a resource, it must execute the modification after the last time the resource was used, in other words, after the free time. If an instruction is going to read the resource, it must perform the read after the last time the resource has been loaded, in other words, after the load time.

The relationship between the temporal usage of each resource and the actual usage of the resource is as follows. If an instruction is going to write/modify the resource, the last time the resource is read or written by other instructions (i.e., the "free time" for the resource) plus one time interval will be the earliest firing time for this instruction. The "plus one time interval" comes from the fact that an instruction is still using the resource during the free time. On the other hand, if the instruction reads a resource, the last time the resource is modified by other instructions (i.e., the load time for the resource) plus one time interval will be the earliest instruction firing time. The "plus one time interval" comes from the time required for the instruction that is performing the load to execute.

The discussion above assumes that the exact location of the resource that is accessed is known. This is always true of resources that are directly named such as general registers and condition code storage. However, memory operations may, in general, be to locations unknown at compile time. In particular, addresses that are generated by effective addressing constructs fall in this class. In the previous example, it has been assumed (for the purposes of communicating the basic concepts of TOLL) that the addresses used by instructions **I0** and **I1** are distinct. If this were not the case, the TOLL software would assure that only those instructions that did not use memory would be allowed to execute in parallel with an instruction that was accessing an unknown location in memory.

The instruction firing time is evaluated by the TOLL software **110** for each resource that the instruction uses. These "candidate" firing times are then compared to determine which is the largest or latest time. The latest time determines the actual firing time assigned to the instruction. At this point, the TOLL software **110** updates all of the resources' free and load times, to reflect the firing time assigned to the instruction. The TOLL software **110** then proceeds to analyze the next instruction.

There are many methods available for determining inter-instruction dependencies within a basic block. The previous discussion is just one possible implementation assuming a specific compiler-TOLL partitioning. Many other compiler-TOLL partitionings and methods for determining inter-instruction dependencies may be possible and realizable to one skilled in the art. Thus, the illustrated TOLL software uses a linked list analysis to represent the data dependencies within a basic block. Other possible data structures that could be used are trees, stacks, etc.

Assume a linked list representation is used for the analysis and representation of the inter-instruction dependencies. Each register is associated with a set of pointers to the instructions that use the value contained in that register. For the matrix multiply example in Table 1, the resource usage is set forth in Table 3:

TABLE 3

Resource	Loaded By	Read By
R0	I0	I2
R1	I1	I2
R2	I2	I3
R3	I3	I3, I2
R4	I4	I5
R10	I0	I0
R11	I1	I1

Thus, by following the "Read by" links and knowing the resource utilization for each instruction, the independencies of Sets 1 and 2, above, are constructed in the analyze instruction stage **120** (FIG. **1**) by TOLL **110**.

For purposes of analyzing further the example of Table 1, it is assumed that the basic block commences with an arbitrary time interval in an instruction stream, such as, for example, time interval T**16**. In other words, this particular basic block in time sequence is assumed to start with time interval T**16**. The results of the analysis in stage **120** are set forth in Table 4.

TABLE 4

REG	I0	I1	I2	I3	I4	I5
R0	T16		T17			
R1		T16	T17			
R2			T17	T18		
R3				T18		
R4					T16	
CC1			T17	T18		
CC2						T17
R10	T16					
R11		T16				

The vertical direction in Table 4 represents the general registers and condition code storage registers. The horizontal direction in the table represents the instructions in the basic block example of Table 1. The entries in the table represent usage of a register by an instruction. Thus, instruction **I0** requires that register **R10** be read and written and register **R0** written at time T**16**, the start of execution of the basic block.

Under the teachings of the present invention, there is no reason that registers **R1**, **R11**, and **R4** cannot also have operations performed on them during time T**16**. The three instructions, **I0**, **I1**, and **I4**, are data independent of each other and can be executed concurrently during time T**16**. Instruction **I2**, however, requires first that registers **R0** and **R1** be loaded so that the results of the load operation can be multiplied. The results of the multiplication are stored in register **R2**. Although, register **R2** could in theory be operated on in time T**16**, instruction **I2** is data dependent upon the results of loading registers **R0** and **R1**, which occurs during time T**16**. Therefore, the completion of instruction **I2** must occur during or after time frame T**17**. Hence, in Table 4 above, the entry T**17** for the intersection of instruction **I2** and register **R2** is underlined because it is data dependent. Likewise, instruction **I3** requires data in register **R2** which first occurs during time T**17**. Hence, instruction **I3** can operate on register **R2** only during or after time T**18**. Instruction **I5** depends upon the reading of the condition code storage CC**2** which is updated by instruction **I4**. The reading of the condition code storage CC**2** is data dependent upon the results stored in time T**16** and, therefore, must occur during or after the next time, T**17**.

Hence, in stage **130**, the object code instructions are assigned "instruction firing times" (IFTs) as set forth in Table 5 based upon the above analysis.

TABLE 5

OBJECT CODE INSTRUCTION	INSTRUCTION FIRING TIME (IFT)
I0	T16
I1	T16
I2	T17
I3	T18
I4	T16
I5	T17

Each of the instructions in the sequential instruction stream in a basic block can be performed in the assigned time intervals. As is clear in Table 5, the same six instructions of Table 1, normally processed sequentially in six cycles, can be processed, under the teachings of the present invention, in only three firing times: T**16**, T**17**, and T**18**. The instruction firing time (IFT) provides the "time-driven" feature of the present invention.

The next function performed by stage **130**, in the illustrated embodiment, is to reorder the natural concurrencies in the instruction stream according to instruction firing times (IFTs) and then to assign the instructions to the individual logical parallel processors. It should be noted that the reordering is only required due to limitations in currently available technology. If true fully associative memories were available, the reordering of the stream would not be required and the processor numbers could be assigned in a first come, first served manner. The hardware of the instruction selection mechanism could be appropriately modified by one skilled in the art to address this mode of operation.

For example, assuming currently available technology, and a system with four parallel processor elements (PEs) and a branch execution unit (BEU) within each LRD, the processor elements and the branch execution unit can be assigned, under the teachings of the present invention, as set forth in Table 6 below. It should be noted that the processor elements execute all non-branch instructions, while the branch execution unit (BEU) of the present invention executes all branch instructions. These hardware circuitries will be described in greater detail subsequently.

TABLE 6

Logical Processor Number	T16	T17	T18
0	I0	I2	I3
1	I1	—	
2	I4	—	—
3	—	—	—
BEU	—	I5 (delay)	—

Hence, under the teachings of the present invention, during time interval T**16**, parallel processor elements **0**, **1**, and **2** concurrently process instructions **I0**, **I1**, and **I4** respectively. Likewise, during the next time interval T**17**, parallel processor element **0** and the BEU concurrently process instructions **I2** and **I5** respectively. And finally, during time interval T**18**, processor element **0** processes instruction **I3**. During instruction firing times T**16**, T**17**, and T**18**, parallel proces-

sor element **3** is not utilized in the example of Table 1. In actuality, since the last instruction is a branch instruction, the branch cannot occur until the last processing is finished in time **T18** for instruction **I3**. A delay field is built into the processing of instruction **I5** so that even though it is processed in time interval **T17** (the earliest possible time), its execution is delayed so that looping or branching out occurs after instruction **I3** has executed.

In summary, the TOLL software **110** of the present illustrated embodiment, in stage **130**, examines each individual instruction and its resource usage both as to type and as to location (if known) (e.g., Table 3). It then assigns instruction firing times (IFTs) on the basis of this resource usage (e.g., Table 4), reorders the instruction stream based upon these firing times (e.g., Table 5) and assigns logical processor numbers (LPNs) (e.g., Table 6) as a result thereof.

The extended intelligence information involving the logical processor number (LPN) and the instruction firing time (IFT) is, in the illustrated embodiment, added to each instruction of the basic block as shown in FIGS. **3** and **4**. As will also be pointed out subsequently, the extended intelligence (EXT) for each instruction in a basic block (BB) will be correlated with the actual physical processor architecture of the present invention. The correlation is performed by the system hardware. It is important to note that the actual hardware may contain less, the same as, or more physical processor elements than the number of logical processor elements.

The Shared Context Storage Mapping (SCSM) information in FIG. **4** and attached to each instruction in this illustrated and preferred embodiment of the invention, has a static and a dynamic component. The static component of the SCSM information is attached by the TOLL software or compiler and is a result of the static analysis of the instruction stream. Dynamic information is attached at execution time by a logical resource drive (LRD) as will be discussed later.

At this stage **130**, the illustrated TOLL software **110** has analyzed the instruction stream as a set of single entry single exit (SESE) basic blocks (BBs) for natural concurrencies that can be processed individually by separate processor elements (PEs) and has assigned to each instruction an instruction firing time (IFT) and a logical processor number (LPN). Under the teachings of the present invention, the instruction stream is thus pre-processed by the TOLL software to statically allocate all processing resources in advance of execution. This is done once for any given program and is applicable to any one of a number of different program languages such as FORTRAN, COBOL, PASCAL, BASIC, etc.

Referring to FIG. **5**, a series of basic blocks (BBs) can form a single execution set (ES) and in stage **140**, the TOLL software **110** builds such execution sets (ESs). Once the TOLL software identifies an execution set **500**, header **510** and/or trailer **520** information is added at the beginning and/or end of the set. In the preferred embodiment, only header information **510** is attached at the beginning of the set, although the invention is not so limited.

Under the teachings of the present invention, basic blocks generally follow one another in the instruction stream. There may be no need for reordering of the basic blocks even though individual instructions within a basic block, as discussed above, are reordered and assigned extended intelligence information. However, the invention is not so limited. Each basic block is single entry and single exit (SESE) with the exit through a branch instruction. Typically, the branch to another instruction is within a localized neighborhood such as within **400** instructions of the branch. The purpose of forming the execution sets (stage **140**) is to determine the minimum number of basic blocks that can exist within an execution set such that the number of "instruction cache faults" is minimized. In other words, in a given execution set, branches or transfers out of an execution set are statistically minimized. The TOLL software in stage **140**, can use a number of conventional techniques for solving this linear programming-like problem, a problem which is based upon branch distances and the like. The purpose is to define an execution set as set forth in FIG. **5** so that the execution set can be placed in a hardware cache, as will be discussed subsequently, to minimize instruction cache faults (i.e., transfers out of the execution set).

What has been set forth above is an example, illustrated using Tables 1 through 6, of the TOLL software **110** in a single context application. In essence, the TOLL software determines the natural concurrencies within the instruction streams for each basic block within a given program. The TOLL software adds, in the illustrated embodiment, an instruction firing time (IFT) and a logical processor number (LPN) to each instruction in accordance with the determined natural concurrencies. All processing resources are statically allocated in advance of processing. The TOLL software of the present invention can be used in connection with a number of simultaneously executing different programs, each program being used by the same or different users on a processing system of the present invention as will be described and explained below.

3. General Hardware Description

Referring to FIG. **6**, the block diagram format of the system architecture of the present invention, termed the TDA system architecture **600**, includes a memory subsystem **610** interconnected to a plurality of logical resource drivers (LRDs) **620** over a network **630**. The logical resource drivers **620** are further interconnected to a plurality of processor elements **640** over a network **650**. Finally, the plurality of processor elements **640** are interconnected over a network **670** to the shared resources containing a pool of register set and condition code set files **660**. The LRD-memory network **630**, the PE-LRD network **650**, and the PE-context file network **670** are full access networks that could be composed of conventional crossbar networks, omega networks, banyan networks, or the like. The networks are full access (non-blocking in space) so that, for example, any processor element **640** can access any register file or condition code storage in any context (as defined hereinbelow) file **660**. Likewise, any processor element **640** can access any logical resource driver **620** and any logical resource driver **620** can access any portion of the memory subsystem **610**. In addition, the PE-LRD and PE-context file networks are non-blocking in time. In other words, these two networks guarantee access to any resource from any resource regardless of load conditions on the network. The architecture of the switching elements of the PE-LRD network **650** and the PE-context file network **670** are considerably simplified since the TOLL software guarantees that collisions in the network will never occur. The diagram of FIG. **6** represents an MIMD system wherein each context file **660** corresponds to at least one user program.

The memory subsystem **610** can be constructed using a conventional memory architecture and conventional memory elements. There are many such architectures and elements that could be employed by a person skilled in the art and which would satisfy the requirements of this system. For example, a banked memory architecture could be used. (*High Speed Memory Systems,* A. V. Pohm and O. P. Agrawal, Reston Publishing Co., 1983.)

The logical resource drivers **620** are unique to the system architecture **600** of the present invention. Each illustrated LRD provides the data cache and instruction selection support for a single user (who is assigned a context file) on a timeshared basis. The LRDs receive execution sets from the various users wherein one or more execution sets for a context are stored on an LRD. The instructions within the basic blocks of the stored execution sets are stored in queues based on the previously assigned logical processor number. For example, if the system has 64 users and 8 LRDS, 8 users would share an individual LRD on a timeshared basis. The operating system determines which user is assigned to which LRD and for how long. The LRD is detailed at length subsequently.

The processor elements **640** are also unique to the TDA system architecture and will be discussed later. These processor elements in one particular aspect of the invention display a context free stochastic property in which the future state of the system depends only on the present state of the system and not on the path by which the present state was achieved. As such, architecturally, the context free processor elements are uniquely different from conventional processor elements in two ways. First, the elements have no internal permanent storage or remnants of past events such as general purpose registers or program status words. Second, the elements do not perform any routing or synchronization functions. These tasks are performed by the TOLL software and are implemented in the LRDs. The significance of the architecture is that the context free processor elements of the present invention are a true shared resource to the LRDs. In another preferred particular embodiment of the invention wherein pipelined processor elements are employed, the processors are not strictly context free as was described previously.

Finally, the register set and condition code set files **660** can also be constructed of commonly available components such as AMD 29300 series register files, available from Advanced Micro Devices, 901 Thompson Place, P.O. Box 3453, Sunnyvale, Calif. 94088. However, the particular configuration of the files **660** illustrated in FIG. **6** is unique under the teachings of the present invention and will be discussed later.

The general operation of the present invention, based upon the example set forth in Table 1, is illustrated with respect to the processor-context register file communication in FIGS. **7***a*, **7***b*, and **7***c*. As mentioned, the time-driven control of the present illustrated embodiment of the invention is found in the addition of the extended intelligence relating to the logical processor number (LPN) and the instruction firing time (IFT) as specifically set forth in FIG. **4**. FIG. **7** generally represents the configuration of the processor elements PE**0** through PE**3** with registers R**0** through R**5**, . . . , R**10** and R**11** of the register set and condition code set file **660**.

In explaining the operation of the TDA system architecture **600** for the single user example in Table 1, reference is made to Tables 3 through 5. In the example, for instruction firing time T**16**, the context file-PE network **670** interconnects processor element PE**0** with registers R**0** and R**10**, processor element PE**1** with registers R**1** and R**11**, and processor element PE**2** with register R**4**. Hence, during time T**16**, the three processor elements PE**0**, PE**1**, and PE**2** process instructions I**0**, I**1**, and I**4** concurrently and store the results in registers R**0**, R**10**, R**1**, R**11**, and R**4**. During time T**16**, the LRD **620** selects and delivers the instructions that can fire (execute) during time T**17** to the appropriate processor elements. Referring to FIG. **7***b*, during instruction firing time T**17**, only processor element PE**0**, which is now assigned to process instruction I**2** interconnects with registers R**0**, R**1**, and R**2**. The BEU (not shown in FIGS. **7***a*, **7***b*, and **7***c*) is also connected to the condition code storage. Finally, referring to FIG. **7***c*, during instruction firing time T**18**, processor element PE**0** is connected to registers R**2** and R**3**.

Several important observations need to be made. First, when a particular processor element (PE) places results of its operation in a register, any processor element, during a subsequent instruction firing time (IFT), can be interconnected to that register as it executes its operation. For example, processor element PE**1** for instruction I**1** loads register R**1** with the contents of a memory location during IFT T**16** as shown in FIG. **7***a*. During instruction firing time T**17**, processor element PE**0** is interconnected with register R**1** to perform an additional operation on the results stored therein. Under the teachings of the present invention, each processor element (PE) is "totally coupled" to the necessary registers in the register file **660** during any particular instruction firing time (IFT) and, therefore, there is no need to move the data out of the register file for delivery to another resource; e.g. in another processor's register as in some conventional approaches.

In other words, under the teachings of the present invention, each processor element can be totally coupled, during any individual instruction firing time, to any shared register in files **660**. In addition, under the teachings of the present invention, none of the processor elements has to contend (or wait) for the availability of a particular register or for results to be placed in a particular register as is found in some prior art systems. Also, during any individual firing time, any processor element has full access to any configuration of registers in the register set file **660** as if such registers were its own internal registers.

Hence, under the teachings of the present invention, the intelligence added to the instruction stream is based upon detected natural concurrencies within the object code. The detected concurrencies are analyzed by the TOLL software, which in one illustrated embodiment logically assigns individual logical processor elements (LPNs) to process the instructions in parallel, and unique firing times (IFTs) so that each processor element (PE), for its given instruction, will have all necessary resources available for processing according to its instruction requirements. In the above example, the logical processor numbers correspond to the actual processor assignment, that is, LPN**0** corresponds to PE**0**, LPN**1** to PE**1**, LPN**2** to PE**2**, and LPN**3** to PE**3**. The invention is not so limited since any order such as LPN**0** to PE**1**, LPN**1** to PE**2**, etc. could be used. Or, if the TDA system had more or less than four processors, a different assignment could be used as will be discussed.

The timing control for the TDA system is provided by the instruction firing times, that is, the system is time-driven. As can be observed in FIGS. **7***a*through **7***c*, during each individual instruction firing time, the TDA system architecture composed of the processor elements **640** and the PE-register set file network **670**, takes on a new and unique particular configuration fully adapted to enable the individual processor elements to concurrently process instructions while making full use of all the available resources. The processor elements can be context free and thereby data, condition, or information relating to past processing is not required, nor does it exist, internally to the processor element. The context free processor elements react only to the requirements of each individual instruction and are interconnected by the hardware to the necessary shared registers.

4. Summary

In summary, the TOLL software **110** for each different program or compiler output **100** analyzes the natural concurrencies existing in each single entry, single exit (SESE) basic block (BB) and adds intelligence, including in one illustrated embodiment, a logical processor number (LPN) and an instruction firing time (IFT), to each instruction. In an MIMD system of the present invention as shown in FIG. **6**, each context file would contain data from a different user executing a program. Each user is assigned a different context file and, as shown in FIG. **7**, the processor elements (PEs) are capable of individually accessing the necessary resources such as registers and condition codes storage required by the instruction. The instruction itself carries the shared resource information (that is, the registers and condition code storage). Hence, the TOLL software statically allocates only once for each program the necessary information for controlling the processing of the instruction in the TDA system architecture illustrated in FIG. **6** to insure a time-driven decentralized control wherein the memory, the logical resource drivers, the processor elements, and the context shared resources are totally coupled through their respective networks in a pure, non-blocking fashion.

The logical resource drivers (LRDs) **620** receive the basic blocks formed in an execution set and are responsible for delivering each instruction to the selected processor element **640** at the instruction firing time (IFT). While the example shown in FIG. **7** is a simplistic representation for a single user, it is to be expressly understood that the delivery by the logical resource driver **620** of the instructions to the processor elements **640**, in a multi-user system, makes full use of the processor elements as will be fully discussed subsequently. Because the timing and the identity of the shared resources and the processor elements are all contained within the extended intelligence added to the instructions by the TOLL software, each processor element **640** can be completely (or in some instances substantially) context free and, in fact, from instruction firing time to instruction firing time can process individual instructions of different users as delivered by the various logical resource drivers. As will be explained, in order to do this, the logical resource drivers **620**, in a predetermined order, deliver the instructions to the processor elements **640** through the PE-LRD network **650**.

It is the context free nature of the processor elements which allows the independent access by any processor element of the results of data generation/manipulation from any other processor element following the completion of each instruction execution. In the case of processors which are not context free, in order for one processor to access data created by another, specific actions (usually instructions which move data from general purpose registers to memory) are required in order to extract the data from one processor and make it available to another.

It is also the context free nature of the processor elements that permits the true sharing of the processor elements by multiple LRDs. This sharing can be as fine-grained as a single instruction cycle. No programming or special processor operations are needed to save the state of one context (assigned to one LRD), which has control of one or more processor elements, in order to permit control by another context (assigned to a second LRD). In processors which are not context free, which is the case for the prior art, specific programming and special machine operations are required in such state-saving as part of the process of context switching.

There is one additional alternative in implementing the processor elements of the present invention, which is a modification to the context free concept: an implementation which provides the physically total interconnection discussed above, but which permits, under program control, a restriction upon the transmission of generated data to the register file following completion of certain instructions.

In a fully context free implementation, at the completion of each instruction which enters the processor element, the state of the context is entirely captured in the context storage file. In the alternative case, transmission to the register file is precluded and the data is retained within the processor and made available (for example, through data chaining) to succeeding instructions which further manipulate the data. Ultimately, data is transmitted to the register file after some finite sequence of instructions completes; however, it is only the final data that is transmitted.

This can be viewed as a generalization of the case of a microcoded complex instruction as described above, and can be considered a substantially context free processor element implementation. In such an implementation, the TOLL software would be required to ensure that dependent instructions execute on the same processor element until such time as data is ultimately transmitted to the context register file. As with pipelined processor elements, this does not change the overall functionality and architecture of the TOLL software, but mainly affects the efficient scheduling of instructions among processor elements to make optimal use of each instruction cycle on all processor elements.

DETAILED DESCRIPTION

1. Detailed Description of Software

In FIGS. **8** through **11**, the details of the TOLL software **110** of the present invention are set forth. Referring to FIG. **8**, the conventional output from a compiler is delivered to the TOLL software at the start stage **800**. The following information is contained within the conventional compiler output **800**: (a) instruction functionality, (b) resources required by the instruction, (c) locations of the resources (if possible), and (d) basic block boundaries. The TOLL software then starts with the first instruction at stage **810** and proceeds to determine "which" resources are used in stage **820** and "how" the resources are used in stage **830**. This process continues for each instruction within the instruction stream through stages **840** and **850** as was discussed in the previous section.

After the last instruction is processed, as tested in stage **840**, a table is constructed and initialized with the "free time" and "load time" for each resource. Such a table is set forth in Table 7 for the inner loop matrix multiply example and at initialization, the table contains all zeros. The initialization occurs in stage **860** and once constructed the TOLL software proceeds to start with the first basic block in stage **870**.

TABLE 7

Resource	Load Time	Free Time
R0	T0	T0
R1	T0	T0
R2	T0	T0
R3	T0	T0
R4	T0	T0
R10	T0	T0
R11	T0	T0

Referring to FIG. **9**, the TOLL software continues the analysis of the instruction stream with the first instruction of the next basic block in stage **900**. As stated previously, TOLL performs a static analysis of the instruction stream. Static analysis assumes (in effect) straight line code, that is,

each instruction is analyzed as it is seen in a sequential manner. In other words, static analysis assumes that a branch is never taken. For non-pipelined instruction execution, this is not a problem, as there will never be any dependencies that arise as a result of a branch. Pipelined execution is discussed subsequently (although, it can be stated that the use of pipelining will only affect the delay value of the branch instruction).

Clearly, the assumption that a branch is never taken is incorrect. However, the impact of encountering a branch in the instruction stream is straightforward. As stated previously, each instruction is characterized by the resources (or physical hardware elements) it uses. The assignment of the firing time (and in the illustrated embodiment, the logical processor number) is dependent on how the instruction stream accesses these resources. Within this particular embodiment of the TOLL software, the usage of each resource is represented, as noted above, by data structures termed the free and load times for that resource. As each instruction is analyzed in sequence, the analysis of a branch impacts these data structures in the following manner.

When all of the instructions of a basic block have been assigned firing times, the maximum firing time of the current basic block (the one the branch is a member of) is used to update all resources load and free times (to this value). When the next basic block analysis begins, the proposed firing time is then given as the last maximum value plus one. Hence, the load and free times for each of the register resources **R0** through **R4**, **R10** and **R11** are set forth below in Table 8, for the example, assuming the basic block commences with a time of **T16**.

TABLE 8

Resource	Load Time	Free Time
R0	T15	T15
R1	T15	T15
R2	T15	T15
R3	T15	T15
R4	T15	T15
R10	T15	T15
R11	T15	T15

Hence, the TOLL software sets a proposed firing time (PFT) in stage **910** to the maximum firing time plus one of the previous basic blocks firing times. In the context of the above example, the previous basic block's last firing time is **T15**, and the proposed firing time for the instructions in this basic block commence with **T16**.

In stage **920**, the first resource used by the first instruction, which in this case is register **R0** of instruction **I0**, is analyzed. In stage **930**, a determination is made as to whether or not the resource is read. In the above example, for instruction **I0**, register **R0** is not read but is written and, therefore, stage **940** is next entered to make the determination of whether or not the resource is written. In this case, instruction **I0** writes into register **R0** and stage **942** is entered. Stage **942** makes a determination as to whether the proposed firing time (PFT) for instruction **I0** is less than or equal to the free time for the resource. In this case, referring to Table 8, the resource free time for register **R0** is **T15** and, therefore, the instruction proposed firing time of **T16** is greater than the resource free time of **T15** and the determination is "no" and stage **950** is accessed.

The analysis by the TOLL software proceeds to the next resource which in the case, for instruction **I0**, is register **R10**. This resource is both read and written by the instruction. Stage **930** is entered and a determination is made as to whether or not the instruction reads the resource. It does, so stage **932** is entered where a determination is made as to whether the current proposed firing time for the instruction (**T16**) is less than the resource load time (**T15**). It is not, so stage **940** is entered. Here a determination is made as to whether the instruction writes the resource. It does; so stage **942** is entered. In this stage a determination is made as to whether the proposed firing time for the instruction (**T16**) is less than the free time for the resource (**T15**). It is not, and stage **950** is accessed. The analysis by the TOLL software proceeds either to the next resource (there is none for instruction **I0**) or to "B" (FIG. **10**) if the last resource for the instruction has been processed.

Hence, the answer to the determination at stage **950** is affirmative and the analysis then proceeds to FIG. **10**. In FIG. **10**, the resource free and load times will be set. At stage **1000**, the first resource for instruction **I0** is register **R0**. The first determination in stage **1010** is whether or not the instruction reads the resource. As before, register **R0** in instruction **I0** is not read but written and the answer to this determination is "no" in which case the analysis then proceeds to stage **1020**. In stage **1020**, the answer to the determination as to whether or not the resource is written is "yes" and the analysis proceeds to stage **1022**. Stage **1022** makes the determination as to whether or not the proposed firing time for the instruction is greater than the resource load time. In the example, the proposed firing time is **T16** and, with reference back to Table 8, the firing time **T16** is greater than the load time **T15** for register **R0**. Hence, the response to this determination is "yes" and stage **1024** is entered. In stage **1024**, the resource load time is set equal to the instruction's proposed firing time and the table of resources (Table 8) is updated to reflect that change. Likewise, stage **1026** is entered and the resource free-time is updated and set equal to the instruction's proposed firing time plus one or **T17** (**T16** plus one).

Stage **1030** is then entered and a determination made as to whether there are any further resources used by this instruction. There is one, register **R10**, and the analysis processes this resource. The next resource is acquired at stage **1070**. Stage **1010** is then entered where a determination is made as to whether or not the resource is read by the instruction. It is and so stage **1012** is entered where a determination is made as to whether the current proposed firing time (**T16**) is greater than the resource's free-time (**T15**). It is, and therefore stage **1014** is entered where the resource's free-time is updated to reflect the use of this resource by this instruction. The method next checks at stage **1020** whether the resource is written by the instruction. It is, and so stage **1022** is entered where a determination is made as to whether or not the current proposed firing time (**T16**) is greater than the load time of the resource (**T15**). It is, so stage **1024** is entered. In this stage, the resource's load-time is updated to reflect the firing time of the instruction, that is, the load-time is set to **T16**. Stage **1026** is then entered where the resource's free-time is updated to reflect the execution of the instruction, that is, the free-time is set to **T17**. Stage **1030** is then entered where a determination is made as to whether or not this is the last resource used by the instruction. It is, and therefore, stage **1040** is entered. The instruction firing time (IFT) is now set to equal the proposed firing time (PFT) of **T16**. Stage **1050** is then accessed which makes a determination as to whether or not this is the last instruction in the basic block, which in this case is "no"; and stage **1060** is entered to proceed to the next instruction, **I1**, which enters the analysis stage at "A1" of FIG. **9**.

The next instruction in the example is **I1** and the identical analysis is had for instruction **I1** for registers **R1** and **R11** as